# Python - Programmierung für die Computerbasierte Intelligenz

Tobias Häuser

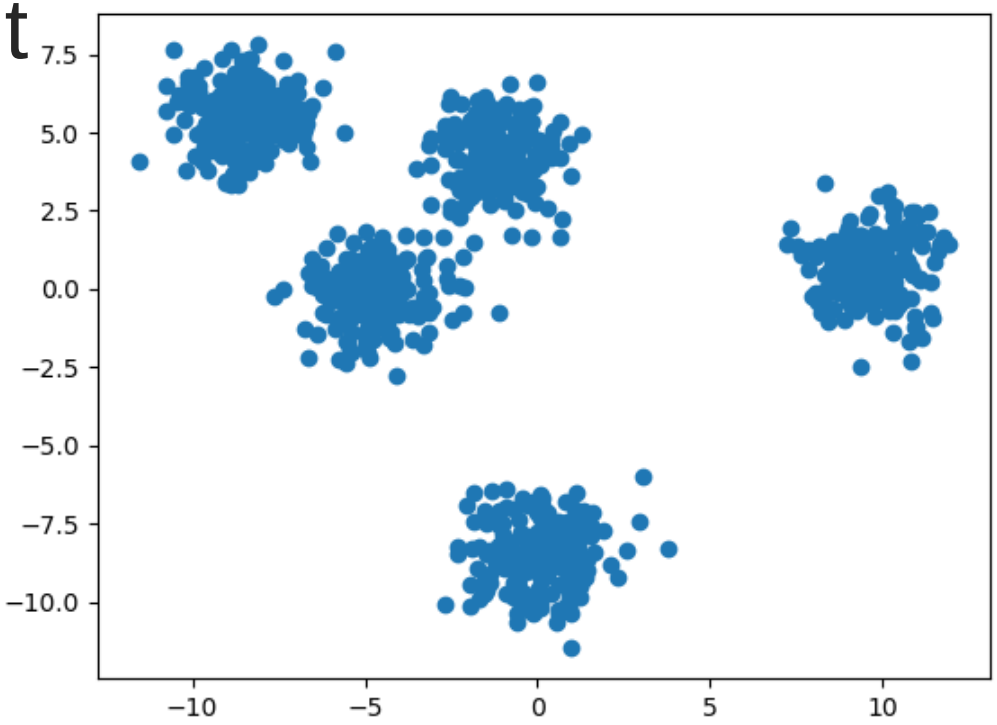
Vorlesung 4.2.

25.08.2018

1. k-Means
    1. Einführung
    2. Eigeneimplementierung
    3. Beispiel Farbenskalierung im Bild
2. Fuzzy k-Means
    1. Einführung
    2. Eigenimplementierung
3. Gustafson-Kessel
    1. Einführung
    2. Eigenimplementierung

# 1.1. k-Means - Einführung

- Ein Algorithmus der zur Clusteranalyse verwendet wird
  - Bzw. sind auf dem Bild 5 Cluster zu erkennen (Gruppen)
  - der PC weiß dies jedoch nicht
  - Wir geben ihm die Anzahl der Gruppen vor und er berechnet welcher dieser Punkte zu welchem Cluster gehören



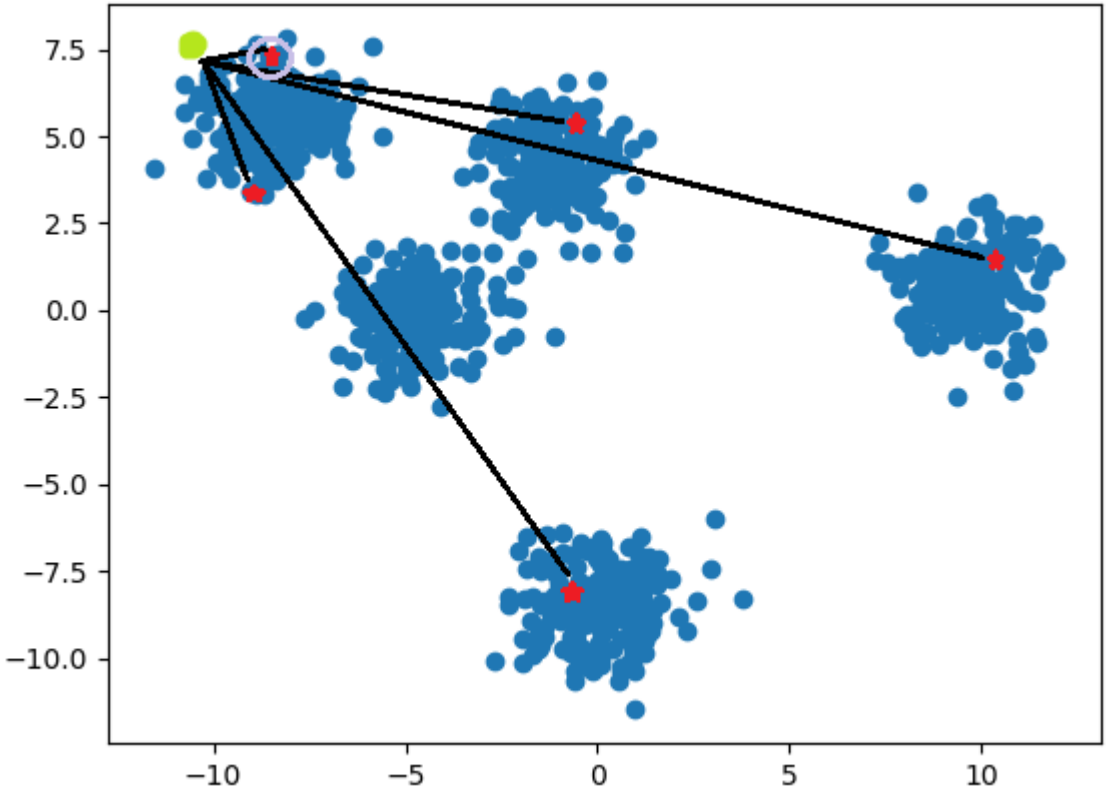
# 1.1. Einführung

- Der am häufigsten verwendete Algorithmus hierbei ist der Lloyd-Algorithmus, welches aus 3 Schritten besteht
  1. Initialisierung der k zufälligen Mittelwerte
  2. Zuordnung der Datenpunkte zu einem Cluster
     - Hierbei wird die euklidische Distanz von jedem Punkt zu jedem Cluster gebildet
     - Der Punkt wird dem Cluster zugewiesen, zu welchem die Distanz am geringsten ist
  3. Berechnen der Mittelpunkte des Clusters

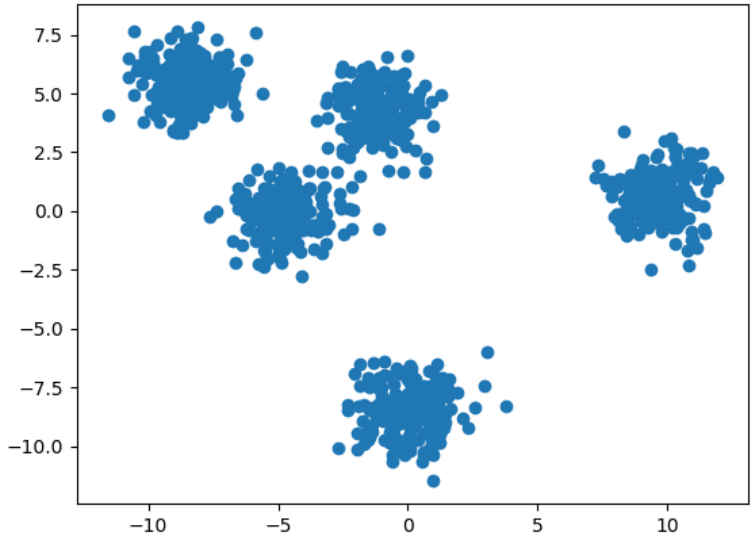
- Schritt 2 & 3 werden wiederholt bis sich die Mittelpunkte nicht mehr ändern

# 1.1. Einführung

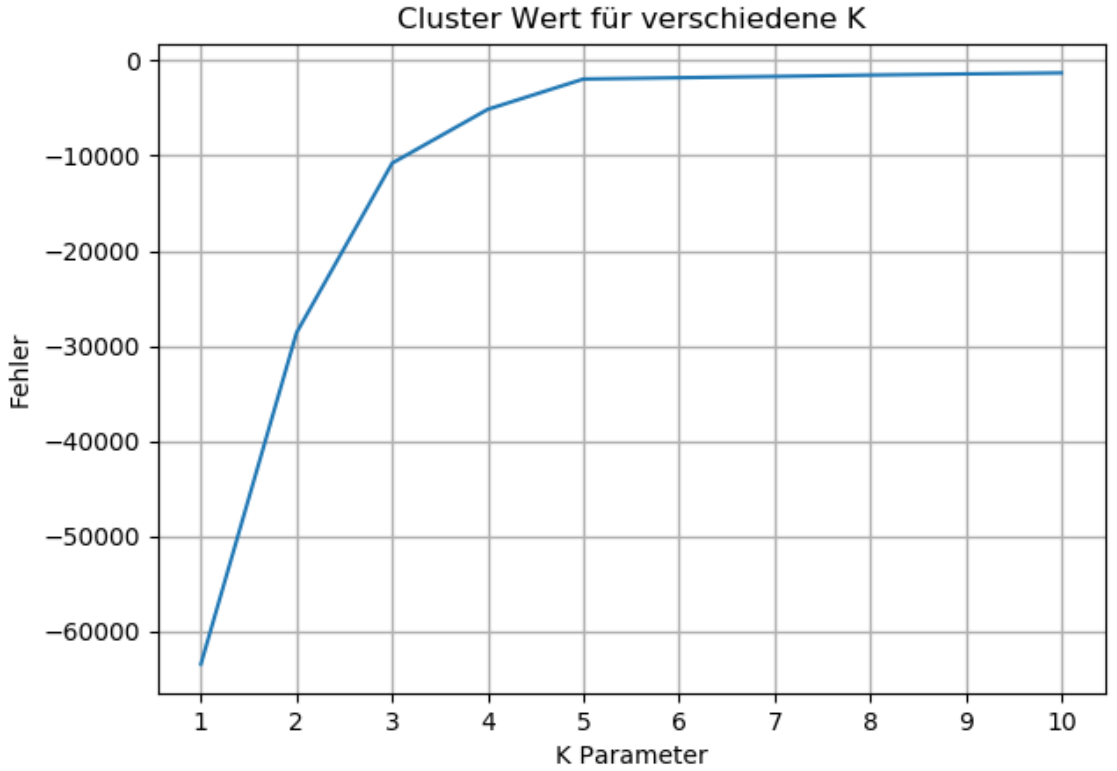
- Grüner Pkt. = zu bestimmender Punkt
- Rote Pkt. = Clustermittelpunkte ($v_i$)
- Distanz = schwarze Linie
- Lila Kreis
  = zugeteiltes Cluster des Punktes



# 1.1. Einführung

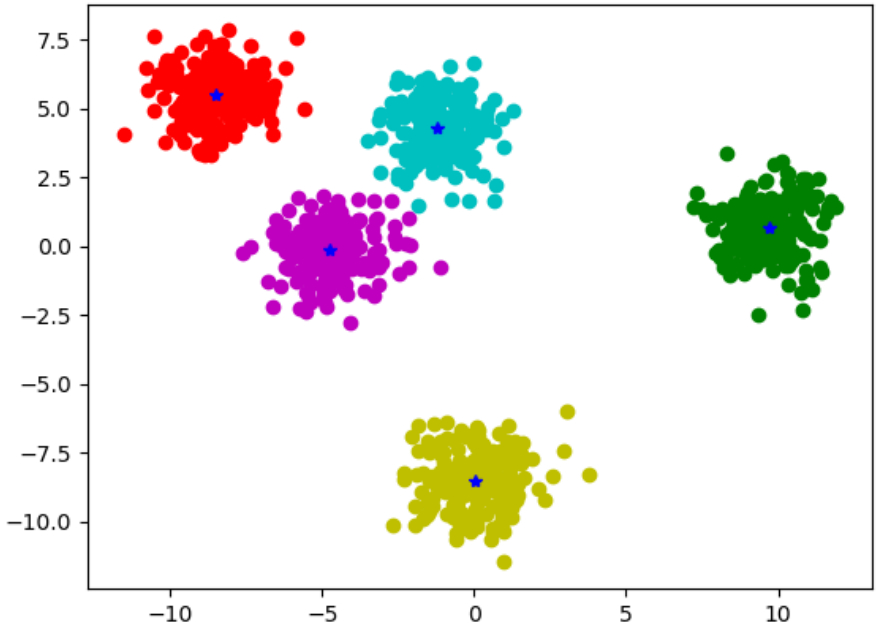



Ausgangsdaten




(Elbow method)

Zieldaten



# 1.2. Eigeneimplementierung

- Für unsere Eigenimplementierung benötigen wir wieder 2 Klassen
  - 1x Main-Klasse & die kMeans Klasse

- In der kMeans Klassen wollen wir 4 öffentliche Funktionen
  - Anzeigen der Ausgangsdaten
  - Anzeigen der optimalen k-Parameter Zuweisung
  - kMeans Berechnung ausführen
  - Anzeigen der zugeordneten Daten

# 1.2. Eigeneimplementierung

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans


# Aufruf zur Erstellung eines Objektes der Klasse
def kmean(data):
    return K_Means(data)


class K_Means:
    # Globale Deklaratio
    __data = []
    __centroids = []
    __labels = []
    __history_centroids = []
    __k = 1
```

# 1.2. Eigeneimplementierung

```
# Initialisierung
def __init__(self, data):
    self.__data = data


# Euklidische Distanzberechnung
def __euclidian(self, a, b):
    return np.linalg.norm(a - b)
```

# 1.2. Eigeneimplementierung

Fkt. Ausgangsdaten plt

```
# Funktion, die zweidimensionalen Daten plottet
def plotData(self, colors=None):
    # X & Y herrausfiltern
    x = self.__data[:, 0]
    y = self.__data[:, 1]

    # Plot, Figure erstellen & ploten
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(x, y, c=colors)
    plt.show()
```

# 1.2. Eigeneimplementierung

Fkt. opti. k-Para. plt

```
# Funktion, die ein Liniendiagramm zeichnet
def plotClusterScore(self):
    # Iteration um passendes K zu finden
    # Ergebnisse plotten
    scores = []
    for i in range(1, 11):
        # kMeans ausführen (sklearn) und jeweiligen Score ermitteln
        print("Clusterberechnung mit K=%d" % (i))
        kmeans = KMeans(n_clusters=i)
        kmeans.fit(self.__data)
        scores.append([i, kmeans.score(self.__data)])

    # Daten in numpy Array umwandeln, X & Y Werte filtern
    scores = np.array(scores)
    x = scores[:, 0]
    y = scores[:, 1]
```

# 1.2. Eigeneimplementierung

```
# Plot, Figure erstellen & ploten
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y, '-')

# Diagrammtitel setzen
plt.title("Cluster Wert für verschiedene K")
plt.xlabel("K Parameter")

# Diagrammeinstellungen % show
plt.ylabel("Fehler")
plt.xticks(x)
plt.grid(linewidth=1)
plt.show()
```

# 1.2. Eigeneimplementierung

Fkt. kMeans ausführen

```
# kMeans ausführen
def kmeans(self, k, epsilon=0):
    # Deklaration
    self.__k = k
    history_centroids = []
    iteration = 0

    # Zeilen & Spaltenanzahl
    num_instances, num_features = self.__data.shape

    # Random Zentralpunkte setzen
    prototypes = self.__data[np.random.randint(0, num_instances - 1, size=k)]
    history_centroids.append(prototypes)
    prototypes_old = np.zeros(prototypes.shape)

    # Leeres Distanzarray
    labels = np.zeros((num_instances, 1))

    # Erste Distanzberechnung zu 0
    norm = self.__euclidian(prototypes, prototypes_old)
```

# 1.2. Eigeneimplementierung

```
# Solange der Fehler zu groß ist
while norm > epsilon:
    iteration += 1

    # Distanz des alter zu den neunen Zentralpunkten
    norm = self.__euclidian(prototypes, prototypes_old)
    prototypes_old = prototypes

    # finden des Zentralpunktes der kürzestens Distanz
    for index_instance, instance in enumerate(self.__data):
        dist_vec = np.zeros((k, 1))

        # Distanzen vom Dataset & Zentralpunkten
        for index_prototype, prototype in enumerate(prototypes):
            dist_vec[index_prototype] = self.__euclidian(prototype,
                                                         instance)

        # Zentralpunkt der kürzestens Dist speichern
        labels[index_instance, 0] = np.argmin(dist_vec)

    # Temp Zentroid
    tmp_prototypes = np.zeros((k, num_features))
```

# 1.2. Eigeneimplementierung

```
    # Bestimmen der neuen Zentralpunkte
    for index in range(len(prototypes)):
        # Clusterpunkte eines Zentroides
        instances_close = [i for i in range(len(labels)) if labels[i] == index]

        # Mittelwertbestimmung der Clusterpunkte (x & y Achsen)
        prototype = np.mean(self.__data[instances_close], axis=0)

        # Zentroid zu Temp hinzufügen
        tmp_prototypes[index, :] = prototype

    # Zentralpunkt setzen
    prototypes = tmp_prototypes
    history_centroids.append(tmp_prototypes)

self.__centroids = prototypes
self.__labels = labels
self.__history_centroids = history_centroids

return prototypes, labels
```

# 1.2. Eigeneimplementierung

Fkt. Zieldaten plt

```
# Plotting der Cluster & Historie
def plotWithCentroid(self):
    # Deklaration
    colors = ['g', 'r', 'c', 'm', 'y', 'k', 'w']
    fig, ax = plt.subplots()
    history_points = []

    # Einfärben der Punkte anhand des Clusters
    for index in range(self.__k):
        # Clusterpunkte herrausfinden
        instances_close = [i for i in range(len(self.__labels)) if self.__labels[i] == index]

        # Punkte einfärben
        for instance_index in instances_close:
            ax.plot(self.__data[instance_index][0], self.__data[instance_index][1], (colors[index] + 'o'))
```

# 1.2. Eigeneimplementierung

```
# Historie-Centroide durchgehen
for index, centroids in enumerate(self.__history_centroids):
    # Centroide des Historie-Centroides durchgehen
    for inner, item in enumerate(centroids):
        # Erster Durchlauf, Pukte zeichnen bzw erstellen
        if index == 0:
            history_points.append(ax.plot(item[0], item[1], 'bo', marker='*')[0])
        # Anderer Durchlauf, Position des Punkte ändern
        else:
            # Position des Punktes ändern
            history_points[inner].set_data(item[0], item[1])
            print("centroids {} {}".format(index, item))

            # Zeit zum zeichnen des t+1 Clusters
            plt.pause(0.3)

# Ende anzeigen
print('Finish')
plt.show()
```

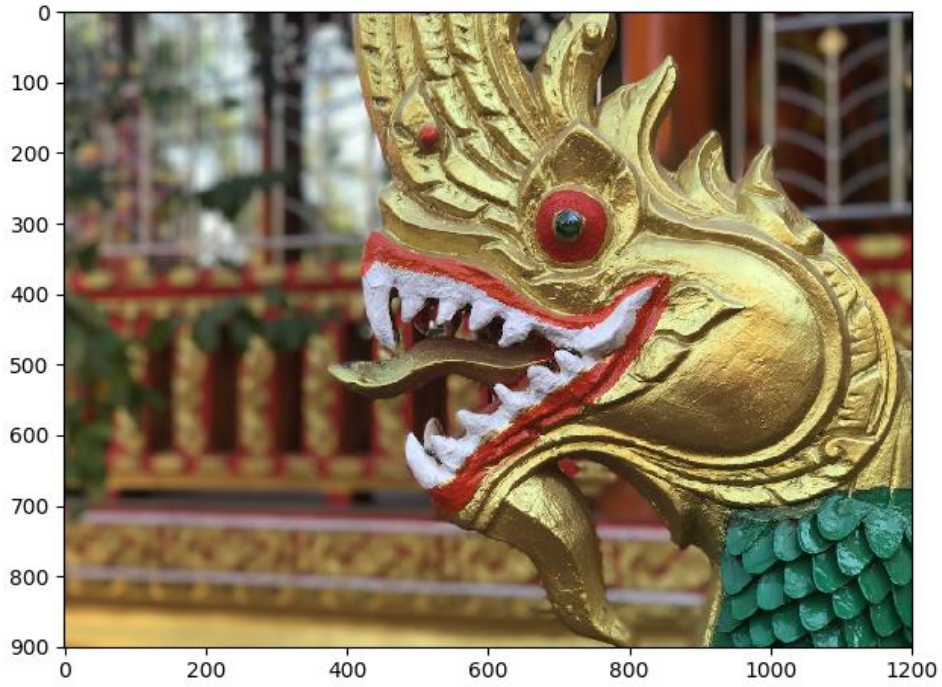
# 1.2. Eigeneimplementierung

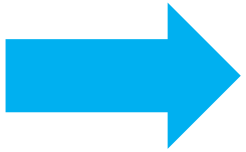
Main Klasse

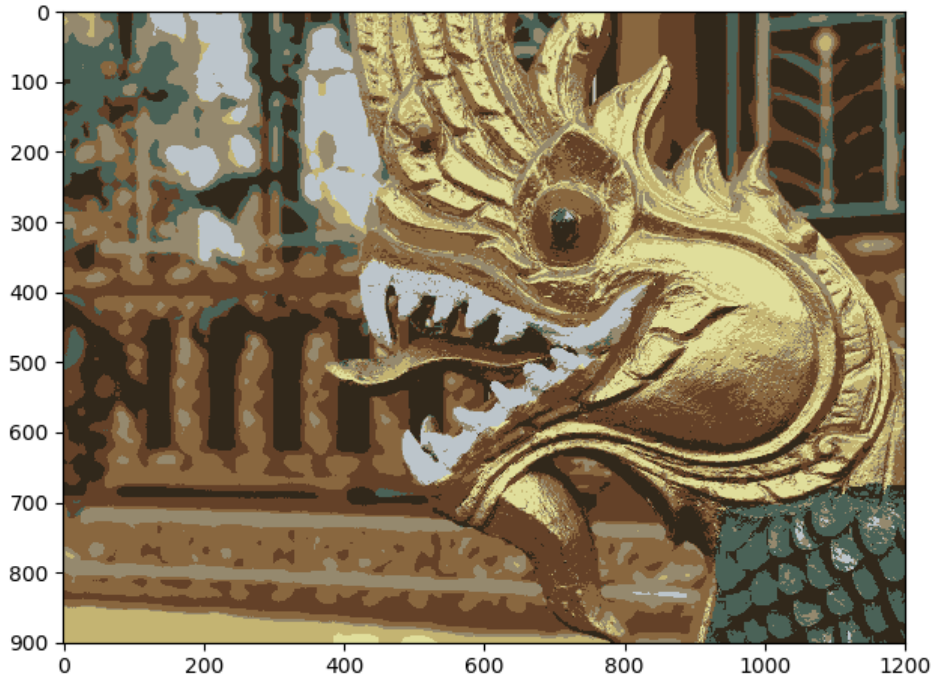
```
from sklearn.datasets import make_blobs
from kmeans import *


# Datensatz generieren
data, classes = make_blobs(n_samples=1000, centers=5, random_state=7)

# Objekt erstellen & Daten übergeben
kmeans = kmean(data)

# Daten ploten (ohne Klassen)
kmeans.plotData()

# Plot der Werte bei verschiedenen Clustergrößen
kmeans.plotClusterScore()

# KMeans ausführen
kmeans.kmeans(5)

# KMeans Plotten
kmeans.plotWithCentroid()
```

# 1.3. Beispiel Farbenskalierung im Bild

- Bild mit vielen Farben bzw. auf nur 8 Farben runter skalieren





# 1.3. Beispiel Farbenskalierung im Bild

Main Klasse

```
from sklearn.cluster import KMeans
from skimage import io
import numpy as np

# Bild
bild = io.imread('dragon.png')[:, :, :3]

# Anzeigen
io.imshow(bild)
io.show()

# Konvert -> 2 Dim Array
data = bild.reshape(-1, 3)

# kMeans Objekt erstellen, Daten übergeben & ausführen
# n_cluster = Anzahl der Farben
model = KMeans(n_clusters=8, n_init=1)
model.fit(data)
```

# 1.3. Beispiel Farbenskalierung im Bild

```
# Centroiden        -> Farben
# Klassifikatoren   -> Pixel
colors = model.cluster_centers_.astype('uint8')
pixels = model.labels_

# Farben anhand der Klassifikation auslesen
pixel2show = colors[pixels]

# Shapen auf 900x1200 Pixel
pixel2show = np.array(pixel2show).reshape(900, 1200, 3)

# Neues Bild anzeigen
io.imshow(pixel2show)
io.show()
```

# 2.1. Fuzzy k-Means - Einführung

- Erweiterung / Abwandlung des k-Means
- Jeder Punkte wird hierbei nicht einem einzelnen Cluster zugeordnet, sondern JEDEM
  - Hierbei wird dem Punkte zu jedem Cluster eine Zuordnung in Prozent berechnet
  - Die Summer der Zuordnungen sind 100%
  - Hierfür kommt eine Zuordnungsmatrix zum Einsatz = $u_{ik}$
    - i = der Punkt
    - k = das Cluster

# 2.1. Einführung

- Als weitere Anpassungsvariable kommt der Fuzzyfier (Verschleifungsgrad) zum Einsatz
  - Einstellung für eine scharfe oder unscharfe Clusterung
  - m sehr groß: Zugehörigkeitswerte $u_{ij}$ liegen ungefähr bei 1/c

    →sehr unscharfe Clusterung(fuzzy)
  - M klein (m=1, …, 2): $u_{ij}$ wird potenziert

    →kleinste Distanzdominiert den Nenner

    →sehr scharfe Clusterung(crisp)

# 2.1. Einführung

- Die Formeln für die Berechnung der Clustercenter ($v_i$) & Zuordnungsmatrix ($u_{ik}$) sind:

$$v_i = \frac{\sum_{k=1}^{N} u_{ik}^{m} x_k}{\sum_{k=1}^{N} u_{ik}^{m}}$$

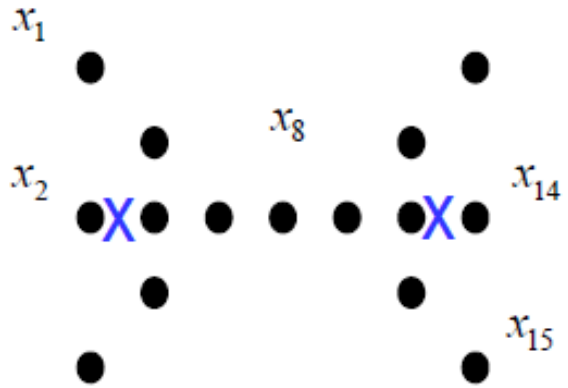
$$u_{ik} = \frac{1}{\sum_{j=1}^{C} \left( \frac{d_{ik}}{d_{jk}} \right)^{\frac{2}{m-1}}}$$

- x = Punkt
- u = Zuord. Pos.
  - i = Pos. des Clusters
  - k = Pos. des Punktes
- m = Fuzzyfier

- d = Distanz
  - j = Pos. des Clusters
  - k = Pos. des Punktes
- m = Fuzzyfier

# 2.1. Einführung

Beispiel:




$$U = \begin{pmatrix} x_1 & x_2 & \dots & x_8 & \dots & x_{14} & x_{15} \\ 0.86 & 0.97 & \dots & 0.5 & \dots & 0.03 & 0.14 \\ 0.14 & 0.03 & \dots & 0.5 & \dots & 0.97 & 0.86 \end{pmatrix} \begin{matrix} c_1 \\ c_2 \end{matrix}$$

- Bsp.

$$u_{11} = \frac{1}{\left(\frac{d11}{d11}\right)^{\frac{2}{2-1}} + \left(\frac{d11}{d21}\right)^{\frac{2}{2-1}}}$$

- $d_{11}$ = Distanz des ersten Punktes zum ersten Clustercenter

# 2.2. Eigenimplementierung

```python
import numpy as np
import matplotlib.pyplot as plt
import random
import math


def fuzzyk(data, m=2):
    return FuzzyK(data, m)


class FuzzyK:
    # Globale Deklaratio
    __data              = []
    __centroids         = []
    __history_centroids = []
    __labels            = []
    __k                 = 1
```

# 2.2. Eigenimplementierung

```
# Fuzzy parameter
__m = 2.00


# Initialisierung
def __init__(self, data, m):
    self.__data = data
    self.__m = m


# Euklidische Distanzberechnung
def __calcEuclidian(self, a, b):
    return np.linalg.norm(a - b)
```

# 2.2. Eigenimplementierung

Init Member Matrix

```
# Initialisierung der MemberMatrix
def __initMemberMatrix(self):
    # leere Liste ertsellen
    memberMat = list()

    # Liste der Punkte und deren Zugehörigkeit initialisieren
    for i in range(len(self.__data)):
        # (0) Zuordnung zu allen -> Random Vollzuordnung (1) zu 1 Centroid
        member = np.zeros(self.__k, dtype=float)
        member[random.randint(0, self.__k - 1)] = 1
        memberMat.append(member)

    return memberMat
```

# 2.2. Eigenimplementierung

Berechn. Centroids

```
# Berechnung aller Centroiden
def __calcClusterCenter(self, membership_mat):
    # Deklarationen
    centroiden = []

    # Berechnung pro Cluster
    for j in range(self.__k):
        # Neu Init
        sumUikXk    = 0
        sumUikYk    = 0
        sumUik      = 0
```

# 2.2. Eigenimplementierung

```
        # Summenfunktionen
        for i in range(len(self.__data)):
            sumUikXk     += membership_mat[i][j] ** self.__m * self.__data[i][0]
            sumUikYk     += membership_mat[i][j] ** self.__m * self.__data[i][1]
            sumUik       += membership_mat[i][j] ** self.__m

        # Berechnung des X & Y Punktes des Centroids
        x = sumUikXk / sumUik
        y = sumUikYk / sumUik

        # Hinzufügen des Centroiden
        centroiden.append([x, y])

    return centroiden
```

# 2.2. Eigenimplementierung

Clusterzuweisung

```
# Aktualisieren der Zugehörigkeiten der Punkte
def __updateMemberMatix(self, membership_mat, cluster_centers):
    # Leeres Distanzarray
    labels = np.zeros((len(self.__data), 1))

    # Jedes Element der MemberMatrix bearbeiten
    for m_idx in range(len(membership_mat)):
        # Jede Zugehörigkeit der k (Cluster) berechnen
        for c_idx in range(self.__k):
            # Neu Init
            sumDikDjk = 0

            # Summenformel des Distanzmaße anhand der Cluster
            for k_idx in range(self.__k):
                # Zähler & Nennerberech des Summenformel
                dik = self.__calcEuclidian(self.__data[m_idx], cluster_centers[c_idx])
                djk = self.__calcEuclidian(self.__data[m_idx], cluster_centers[k_idx])
```

# 2.2. Eigenimplementierung

```
            # Teilen durch 0 vermeiden
            if dik != 0:
                # Quotient der Distanzen hoch... Beachtung des Fuzzyness Parameters
                sumDikDjk += (dik / djk) ** (2 / (self.__m - 1))
            else:
                sumDikDjk = float('nan')

        # Laut Umstellung Fuzzy Formel
        sumDikDjk = sumDikDjk ** -1

        # Hinzufüger der Clusterzuordnungen
        if math.isnan(sumDikDjk):
            membership_mat[m_idx][c_idx] = 1
        else:
            membership_mat[m_idx][c_idx] = sumDikDjk
```

# 2.2. Eigenimplementierung

```
    # Centroid der maximalen Clusterzuordnung
    labels[m_idx, 0] = np.argmax(membership_mat[m_idx])

# Selfvariable übergeben
self.__labels = labels

return membership_mat
```

# 2.2. Eigenimplementierung

Fuzzy kMeans

```
# FuzzyCMeans ausführen
def fuzzyCMeansClustering(self, k, epsilon=0.01, max_iteration=100, updateMat=None):
    # Initialisierung
    self.__k = k
    norm = 1
    iteration = 0
    cluster_centers_old = np.zeros(k * 2)
    cluster_centers = []

    # Init Funktion
    if updateMat is None:
        updateMat = self.__updateMemberMatix

    # Member Matrix init
    member_mat = self.__initMemberMatrix()
```

# 2.2. Eigenimplementierung

```
# Differenz der alten & neuen Cluster zum Epsilonwert
while norm > epsilon or iteration > max_iteration:
    # Erhöhung der Interation
    iteration += 1

    # Centroid & MemberMatrix Berechnung
    cluster_centers = self.__calcClusterCenter(member_mat)
    member_mat      = updateMat(member_mat, cluster_centers)

    # Differenz der alten, neuen Cluster
    # Save new into old
    # ravel() in 1 Zeile konvertieren
    norm = self.__calcEuclidian(np.array(cluster_centers).ravel(), cluster_centers_old)
    cluster_centers_old = np.array(cluster_centers).ravel()
```

# 2.2. Eigenimplementierung

```
    # Historie Centroid Konvertierung für Ploting
    centroid = []
    for idx in range(k):
        centroid.append(cluster_centers[idx])
    self.__history_centroids.append(centroid)

self.__centroids = cluster_centers

return self.__centroids, self.__labels
```

# 2.2. Eigenimplementierung

Main Klasse

```
from sklearn.datasets import make_blobs
from fuzzykmeans import *

# Datensatz generieren
data, classes = make_blobs(n_samples=1000, centers=5, random_state=7)

# Objekt erstellen & Daten übergeben
fuzzyK = fuzzyk(data)

# Fuzzy kMeans ausführen
fuzzyK.fuzzyCMeansClustering(5)
#fuzzyK.GK_CMeansClustering(5)

# Ploting
fuzzyK.plot()
```

# 3.1. Gustafson-Kessel - Einführung

- Erweiterung des Fuzzy k-Means
- Eigenschaften
  - Repräsentiert hyperkugelförmige und hyperellipsoidale Clusterformen
  - Falls $\boldsymbol{C}$ schlecht konditioniert

    →numerische Probleme bei Invertierung
  - Konvergenzprobleme: lokale Minima in $J_m$

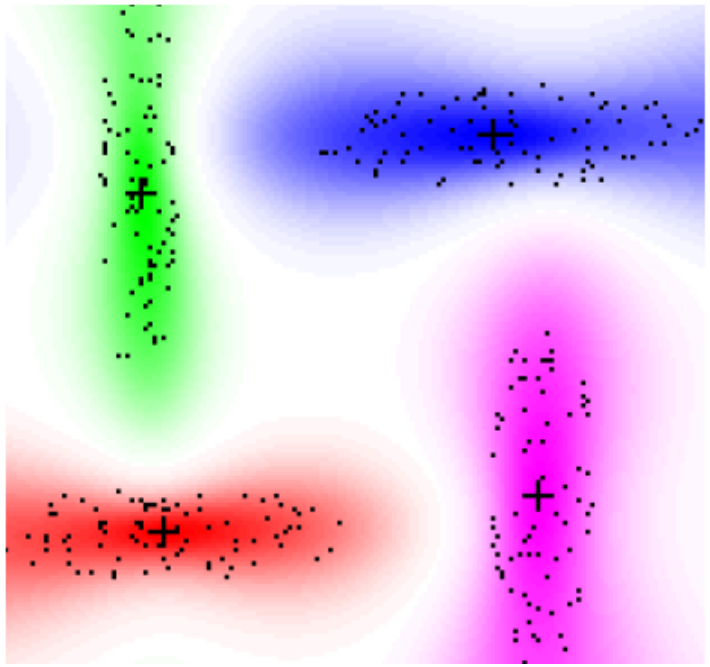
# 3.1. Einführung

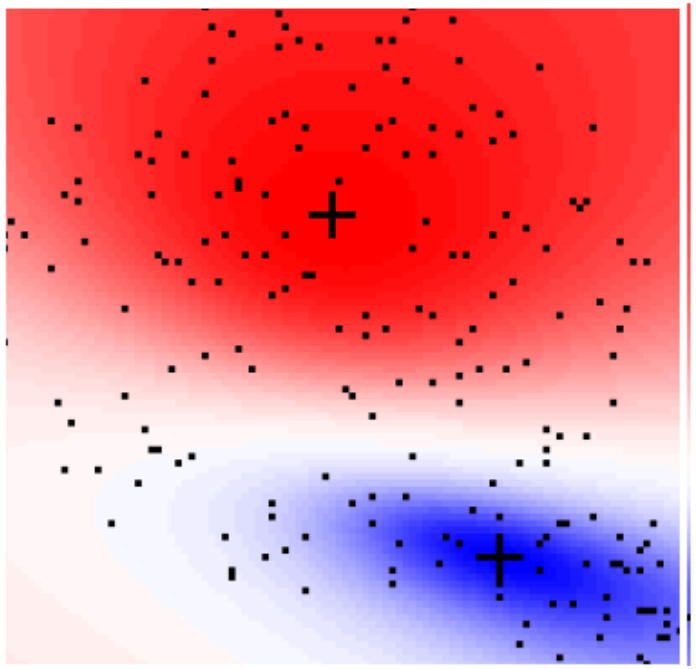
**Distanzmaß:**

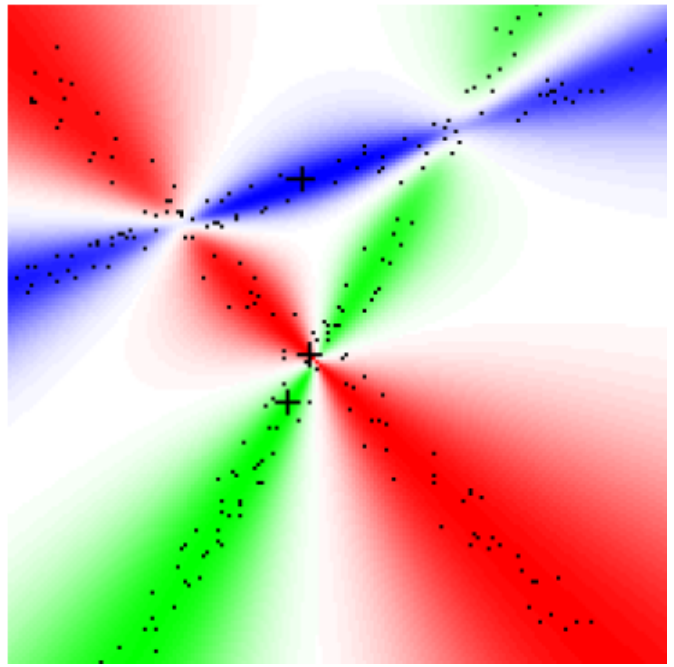
Skalierte Mahalanobis Distanz $\left\|\boldsymbol{x}_j - \boldsymbol{v}_i\right\|^2 = \sqrt[n]{\det \boldsymbol{C}_i}\,(\boldsymbol{x}_j - \boldsymbol{v}_i)^T \boldsymbol{C}_i^{-1}(\boldsymbol{x}_j - \boldsymbol{v}_i)$

(Skalierung mit Streuvolumen)

Fuzzy-Kovarianzmatrizen $\boldsymbol{C}_i = \frac{1}{N_i}\sum_{j=1}^{n}(u_{ij})^m(\boldsymbol{x}_j - \boldsymbol{v}_i)(\boldsymbol{x}_j - \boldsymbol{v}_i)^T ; N_i = \sum_{j=1}^{n}(u_{ij})^m$







# 3.2. Eigenimplementierung

Mahalanibis Dist.

```
# Skalierte Mahalanobis Distanzberechnung
def __calcMahalanobis(self, datepoints, v, ci):
    # Konvertieren un np.Array
    datepoints = np.array(datepoints)
    v = np.array(v)

    # Shapen damit nur Zeilenvektoren vorliegen
    # vi = Zeilenvektoren aller Cluster als Matrix -> Diff ist Matrix zw. xi und v(alle)
    xj = datepoints.reshape(datepoints.shape[0], 1, -1)
    vi = v.reshape(1, v.shape[0], -1)

    # Maritzen subtrahieren und um 1 Dim erhöhen -> Spaltenvektorbildung
    dif_xj_vi = np.expand_dims(xj - vi, axis=3)

    # Determinate bilden und Wurzel ziehen
    determ = np.power(np.linalg.det(ci), 1 / self.__m)
```

# 3.2. Eigenimplementierung

```
# Determinate Reshapen, damit alle Cluster einzeln und mal der Inverse von Ci gerechnet werden kann
detCi_CiInvers = determ.reshape(-1, 1, 1) * np.linalg.inv(ci)

# Transpose der 2 & 3 Dim -> aus Spalten- wird Zeilenvektor
# Matrizen multiplikation
temp = np.matmul(dif_xj_vi.transpose((0, 1, 3, 2)), detCi_CiInvers)

# MatMultiplikation, auf 2 Dim runterbrechen & Transponieren
output = np.matmul(temp, dif_xj_vi).squeeze().T

# Rückgabe des größeren Wertes
return np.fmax(output, 1e-8)
```

# 3.2. Eigenimplementierung

Kovarianz Matrix

```
# Fuzzy Kovarianzmatrizen bilden
def __covariance(self, dataPoints, v, u):
    # Umwandeln in np.Array & hoch m rechnen
    um = np.array(u).transpose() ** self.__m
    v = np.array(v)

    # Bilden der Summe Ni & der Diff xj zu vi
    ni = um.sum(axis=1).reshape(-1, 1, 1)
    xj_vi = dataPoints.reshape(dataPoints.shape[0], 1, -1) - v.reshape(1, v.shape[0], -1)

    # Umwandlung in Spaltenvektoren pro Cluster
    # expand_dims = erhöht die Dimensionen um 1
    # axis=3 Bezieht sich auch die dritte Achse die als neue Dimension dient
    # in dem Fall 4 Dimensionen da 1 Zeilenvektor zu 2 Dim. in Spaltenvektor is
    temp = np.expand_dims(xj_vi, axis=3)
```

# 3.2. Eigenimplementierung

```
# Transponieren der 3 mit 4 Dimension
# Multiplizieren der Vektoren
temp_trans = temp.transpose((0, 1, 3, 2))
temp = np.matmul(temp, temp_trans)

# Transponieren, erhöhen um 2 Dimensionen für die Multiplikation mit Matrix
sum_fkt = um.transpose().reshape(um.shape[1], um.shape[0], 1, 1) * temp

# Summe über 0 Dimension, bzw. summieren/zusammenfassen über jedes Cluster
# Dim 0 = 1 DS für je dataPoint -> Summe bilden bezügl. Summenfkt
sum_fkt = sum_fkt.sum(0)

return sum fkt / ni
```

# 3.2. Eigenimplementierung

Clusterzuweisung

```
# Aktualisieren der Zugehörigkeiten der Punkte
def __updateMemberMatix_GK(self, member_mat, cluster_centers):
    # Leeres Distanzarray
    labels = np.zeros((len(self.__data), 1))

    # Fuzzy Kovarianzmatrizen bilden
    f_dik = self.__covariance(self.__data, cluster_centers, member_mat)

    # Distanzwerte
    # Zähler & Nennerberech des Summenformel
    dist = self.__calcMahalanobis(self.__data, cluster_centers, f_dik)

    # MemerMat updaten
    member_mat = np.transpose(self.__build_u(dist))

    # Centroid der maximalen Clusterzuordnung
    for idx in range(len(member_mat)):
        labels[idx] = np.argmax(member_mat[idx])

    # Selfvariable übergeben
    self.__labels = labels

    return member_mat
```

# 3.2. Eigenimplementierung

GK aufrufen

```
# FuzzyCMeans ausführen
def GK_CMeansClustering(self, k, epsilon=0.01, max_iteration=100):
    return self.fuzzyCMeansClustering(k, epsilon=epsilon, max_iteration=max_iteration,
                                      updateMat=self.__updateMemberMatix_GK)
```

- Für die Main Klasse nutzen wir die vorherige aus dem Fuzzy k-Means und ändern die Zeile für Fuzzy k-Means ausführen zu GK Aufruf

# Literaturhinweise

- Folien aus der Vorlesung CI1
  - CI1_05_Klassif-&Grupp-Analyse_3_Fuzzy-Cluster-Analyse
- http://w3.ualg.pt/~jvo/ml2015-16/ml2015-16L17.pdf
- https://de.wikipedia.org/wiki/Fuzzy_C-Means
- https://blog.ancud.de/home/-/blogs/einfuhrung-in-machine-learning-mit-python-k-means-clusteri-1